(1960s)

GRAPHICAL COMMUNICATION AND CONTROL LANGUAGES

by

L. G. Roberts

Lincoln Laboratory,* Massachusetts Institute of Technology
Lexington, Massachusetts

September 1964

---

* Operated with support from the U. S. Air Force.

MS-1173

PREPRINT

GRAPHICAL COMMUNICATION AND CONTROL LANGUAGES

by

L. G. Roberts

ABSTRACT

The on-line use of a display scope and light pen to control and communicate with computer programs or to construct and manipulate data files can be a very powerful means of communication. Moreover, man-machine communication using a graphical language provides extra dimensions of flexibility beyond the single dimension of typewriter language. In order to provide graphical languages, it has been found that complex list structures must be used. The specific requirement is that these structures be capable of providing multi-dimensional associations. Since no present list language is capable of providing these associations conveniently, a new list structure system and macro language, "CORAL," has been developed for Lincoln Laboratory's TX-2 Computer.

The list structure generated by CORAL utilizes variable length blocks of storage each of which contains one or more ties to other blocks. Beyond some simple list operators, the CORAL language provides operators for conditional testing, algebraic data manipulation, and for creating ordered class associations.

Graphical construction techniques can be used to create both programs and data once the necessary graphical routines and display hardware have been developed. An example of how a user could manipulate a graphical flow chart compiler to construct a graphical electric circuit simulator is presented.

INTRODUCTION

On-line graphical control and display of computer processes has proved extremely effective in certain problem areas for improving the ease and speed of man-machine communication. Although the most significant applications so far have been to graphical problems, it is very likely that most problem-solving activities could benefit significantly from the use of graphical techniques. A graphical language is tremendously powerful because it is a natural form of human representation and it derives richness and economy from its multi-dimensional character.

At Lincoln Laboratory several graphical systems have been implemented, taking advantage of the TX-2 Computer's[1] on-line operation and display capability. Display units have been utilized for graphic outputs ranging from text and graphs to intensity modulated motion pictures and rotating three-dimensional objects. The first significant graphical construction program to demonstrate the powerful input capability of a display scope and light pen was Sketchpad.[2] This system provides the ability to draw pictures and define subpictures in such a way that the computer contains complete information about the topology and structure which was defined. Although Sketchpad itself cannot do much more than produce drawings, the internal list structure which it creates is a computer representation of the two-dimensional construction which can be exploited by other computer programs as an input medium. Sketchpad III[3] extends the graphical construction process to three-dimensions by enabling one to draw in any of three views on the scope thus constructing a computer representation of a three-dimensional drawing.

The effectiveness of graphical languages depends on the creation of graphical utility packages with which other computer programs can easily communicate. Programs such as Sketchpad have demonstrated techniques for the manipulation and creation of drawings but, the real future for graphics lies in being able to manipulate data files and programs external to the graphical package. The pictures are really abstractions used as labels for the external entities so that it is possible to create, interconnect, and rearrange the entities with a two-dimensional language rather than the normal one-dimensional text stream.

The manipulation of component parts in a picture or drawing may seem simple on the surface but, in order to maintain the interrelations between the components a complex storage structure is needed. If some connection is deleted it is not sufficient to just remove the line from a display file. That line represents an association between two elements and may constrain the movement of the elements, the distance between them and their deletion, as well as their external properties (e.g., electrical, program flow, etc.). Thus, each line or other picture element must be attached to an undetermined number of graphical elements and constraints in such a way as to facilitate the processing of these associations.

List Structures

The immediate problem when considering manipulating data structures graphically is that of dimensionality. The internal data structure used by Sketchpad and other graphical systems is by necessity a multi-dimensional form similar to an interconnected list structure, and, in most cases, it would be advantageous if the external data structure were multi-dimensional also. One solution is for the external programs to use the same list structure storage and language as the graphical routines. For example, in a flow chart compiler both the compiler and the Sketchpad-like construction program would work from the same list structure representation of the program flow chart.

Since the list structure utilized in any graphical effort will have to be used outside the graphical packages, as well as inside, there is a need for a fairly complete and well documented list structure language. The well-known list structure languages such as LISP were not designed for graphics and are not very efficient or easy to use for such multi-dimensional problems. They are well suited for processing strings of text but break down when two-way associations between list elements are the rule rather than the exception.

We have tried to develop a list structure system which is efficient and useful for graphical problems as well as for compiling, simulation, and other problem areas. The basic list structure concepts are similar to those used to implement Sketchpad, Sketchpad III, and other graphical projects on the

TX-2 in the past.[4] However, the storage space required has been reduced and a more complete list structure system and language generated.

CORAL

This list structure processing system and language being developed at Lincoln is called CORAL (Class Oriented Ring Association Language). The language consists of a set of operators for building, modifying, and manipulating a list structure as well as a set of algebraic and conditional forms. Presently, the operators are implemented with macro definitions for the TX-2's macro assembler, Mark IV. The operator macro names are compound characters constructed from the extensive character set available on TX-2's Lincoln Writers. Since Mark IV is capable of compiling compound statements and allows a flexible set of operator symbols and terminators, the present CORAL statements would be modified little were a compiler available. The main difference is that Mark IV requires many more parentheses than would otherwise be necessary.

Basic Ring Structure

CORAL list ties are always formed as rings. Each element in a ring requires one 36-bit word and contains a 17-bit pointer to the next element. One element of the ring is the start or head and all the other elements are subordinate to it. The ring start element contains, besides its forward pointer, 9 bits of data and a 9-bit identification code (-0) which marks it as a start point. Every other element of a ring has a second 17-bit pointer which either points to the ring start element or is used as a backward pointer. One bit marks which type of pointer is being used and rings are built with back pointers and start pointers alternating. Back pointers point to the closest previous element with a back pointer so that they form a complete ring. Alternation of the less useful pointer types retains the advantages of both pointers in half the space (one word per element) and with only a small loss of time. (Figure 1 illustrates the basic ring structure.)

Blocks

A block of elements collects many ties together and thus allows the multi-dimensional associations required for graphical data structures.

Although it is sufficient to use at most two ring elements per block, more are often used for increased efficiency. A block is formed from a sequence of registers of any length and contains a blockhead identifier at the top, a group of ring elements and any number of data registers. (Figure 2 illustrates an example of a block.) Blocks are used to represent items or entities and the rings form associations between blocks. Thus, an element may be reached by moving up or down in a block rather than going around its ring. To find out what group the element belongs to it is necessary to find the head element of its ring and this is made efficient through the use of the start ties. If it is necessary to delete an element or insert a new element before it, the back ties are used to find the previous element. Thus, the full set of ties are necessary in the list elements if deletion, insertion, and identifying the elements' group are to be accomplished efficiently.

For example, as illustrated in Figure 3, a triangle may be represented by three point blocks and three line blocks. Each point block has data storage for its X-Y coordinates and has a ring start element with the ring going through all the lines connected to the point. Each line block has two ring elements, one in each of two point rings, as well as a ring element used to group all the lines of the picture together. To display the triangle, one moves around the ring of all lines and at each line block moves down to the end point ties, then moves to their ring starts to find the point blocks and thus the coordinate values.

Free Storage

The list structure storage allocation system is assigned some area of memory and slowly utilizes this reagion as blocks are requested. Since all ties are complete rings rather than one-way pointers, the system subroutines always know when a block has had all associations removed from it and thus cannot be found again by the user. When this occurs or when the user designates that he is through with a block, the block is tied onto the free storage lists. The blocks are grouped according to their length on several lists so that when a certain size block is requested, a block of that size can be retrieved. If no block is available of the right size then the largest free block is split up or if necessary, a new section is cut off the remaining

free memory. Every block is identifiable by type by examining the top register. Thus, before a new free block is added to the free lists, the block just after it in memory is examined and if it is free also, then the two blocks are merged into one larger free block. This merging technique makes the allocation of variable length blocks almost as efficient as the allocation of fixed length blocks. It has been determined experimentally, by testing the actual system on blocks of random length which were either created or destroyed with equal probability, that total storage space required is only about 12 per cent greater than actually used. In some runs as many as 10,000 blocks were created and deleted from a structure containing only 1600 blocks without exceeding this percentage.

Class Structures

The basic list operations considered above, are really just the foundation of CORAL. The sophisticated user would like to be able to talk about items and associations and not have to plan all the ring starts, ties and search procedures he might need. Therefore, a set of class operations have been included in CORAL. The nature of the class properties can best be understood by considering each item as a point in space and associations between items as one-way arrows between points. Each item has a 17-bit type number and perhaps further data and these numbers can be considered as labels on the points. A statement may be made to put item A in class B and this produces an arrow from B to A.

Put in Class: A ⊂ B

As many "put" statements may be used as the user wishes and normal rings are built until an element must belong to two rings. Then a special type of interconnecting item called a nub is introduced to tie two rings together. A nub is a two-word block with both words being ring elements. The extra bit in the ring element is used to indicate that the block is an element pair and has no blockhead identifier.

There are two operators for moving around classes, one to go through all members of a class (arrows leaving this item), and one to find all the

classes an item belongs to (arrows pointing at this item). The operators have a right hand parameter, either a subroutine name or a statement which is executed for each member of the class.

Go Round Class: B ⊃ SB B ⊂ SB

These class "go round" statements can be used in a nested or recursive form. The sequence in which associations were made is maintained by the ring structure so that sequential information as well as associative information is provided. Actually, two forms of the class "put" operator exist so that an element may be put first or last in a class.

Although the sequential ordering of class members may be sufficient in some cases, it is often useful to order classes numerically by some data available from the items in the class. Sometimes all that is wanted is to divide the class into a set of discrete subgroups such as the groups of different type blocks. In other cases, it is desirable to order the sequence of items in a class by some continuous numerical property like their priority value. In either case it is important to be able to search for a particular subset of the class without wasting time checking the other items. When an ordering criterion is provided, the list structure can be organized more like a tree than a list so that fast searches for subgroups can be made.

The CORAL class operators described previously can be used to create ordered classes and search them, simply by stating additional parameters to specify the ordering criteria. All ordering is done using 36-bit numbers which should be obtainable from the items in the class. The user writes a subroutine which will return with the ordering number when given a pointer to one of the items. The subroutine approach leaves the ordering criteria very flexible since the numbers need not be stored directly in the items; they may be found in associated blocks in any other way. Even though a 36-bit number is required for ordering, any smaller data word can be used merely by sign-extending it to full length. The ordering is accomplished using a normalized form of the number so that the significant bits need not be at the front of the word. To create an ordered class, a computation subroutine or statement (SC) is used to condition the class name (B) as new items (A)

are put in the class.

Put In Ordered Class: A ⓒ B|SC

Going around the whole class utilizes the same statement as before except that now the members are found in the sequence dictated by the ordering numbers, from smallest to largest. Only within a subgroup with the same number will the generation order be important. If a particular subgroup of the class is all that is wanted, the number for the requested subgroup is picked up by one parameter (X) and compared with the number generated by the subroutine (SC). The items are only mentioned to the statement subroutine (SB) if the numbers agree.

Search Ordered Class: B|X=SC ⊐ SB

The search for the correct subgroup is made efficient by introducing into the association ties during their creation, additional interconnecting blocks, usually of the element pair variety. These blocks have data stored in them and act like tree branch points to direct the search down one ring or another. Larger blocks are used for branch tables whenever there are enough elements in the class to make such a table useful. None of these decision blocks are used until more than four elements are put in a class since no advantage is gained. The addition and deletion of decision blocks is automatic and dictated by the number and distribution of class members. Simple checks and tree prunning algorithms are used whenever associations are changed and after a number of changes, a complete overhaul of the tree is made. The overall effect of these procedures still requires further evaluation to obtain accurate numbers but, in general, the storage added by the decision blocks is less than 25 per cent and the search speed is improved from a linear to logarithmic function of the number of associations. In fact, when branch blocks are used the search time becomes faster than logarithmic.

A simple example of an ordered class utilizes ordering based on the type number of the blocks. Each type of block such as a picture, point or line has a particular type number which is found for any block by the operator

T . Thus, if we have picture blocks which have other pictures as well as lines as subparts, either type can easily be separated out of the class. Given a picture P, we can find all the lines by stating:

P | LINE = (T) ⊃ SB

This requests the pointers to all the blocks with the type LINE in the class P to be considered by subroutine SB.

It is difficult to emphasize strongly enough the importance of being able to keep adding associations to a fixed size block and at the same time to be able to label these associations so that any desired group of associations may be found. The class ordering accomplishes this and without making the search time unreasonably long. The list structure blocks in Sketchpad have about twice as many ring tie elements as would be necessary with CORAL simply because ties had to be provided whenever transient associations might be needed. Along with the basic ring element size reduction, this means that if Sketchpad were rewritten with CORAL the list storage could be reduced to one fourth its original size.

Deletion

There are two delete operators in CORAL, one for removing associations and the other for deleting items. The removal of an association is very similar to creating one. To remove A from class B one writes:

Delete Association: A ⊗ B

The result is just to remove the arrow between A and B. The delete routines also remove any interconnecting blocks which become unnecessary and, if a tree exists it may be reorganized. All the associations tied to a particular ring element (B) can be deleted by stating:

Delete All Associations: B ⊗

In this case all the ties are removed, the unnecessary interconnecting blocks are deleted, and all subordinate blocks are deleted. For the class statements given previously, all the members of the class B (items with an arrow from B) would be subordinate and hence be deleted. However, CORAL does allow the class ordering techniques to be used independently of the subordinate-superior deletion property. There are slightly different forms for the class statements which create and search a class of superior items.

The deletion of an item or block deletes all the associations from the block specified, and then returns the block to free storage. The delete routines always check any block which has an association deleted from it and if there are no associations left connected to the block it is returned to free storage, thus preventing unattached blocks from collecting. The deletion of subordinate blocks allows one to organize his superiorities such that when any item is deleted, all items which depend on it are deleted. For example, a line depends upon its endpoints and the deletion of a point should cause all the lines which terminate on it to be deleted also. Thus, the point should be made superior to the line.

Master Blocks and Properties

The whole list structure starts with a master-master block which contains the class of all free blocks ordered by length as well as the class of all master blocks ordered by type number. Master blocks are defined by the user for each type block he needs and contain the properties of that type of block. The most basic property is how to build the blocks. The block length is variable for any one type, however, a normal length is given in the master block. Also, the number of tie elements is given and, if specified, the purpose of each tie. Further, the master block property class has the important function of specifying how this type of block should be treated by other programs. For example, the display service program would expect to find the display property of the block as well as the method of constructing such blocks. One last function of the master blocks is to hold the first level or normal blocks, so that all blocks can be reached from the master-master. For example, the class of all pictures might be tied to the picture block master and since the lines tie to the pictures they would not

need to be tied to the line master.

On-Line Problem-Solving Using Graphical Techniques

An example best illustrates the kind of on-line problem-solving capability which a fully developed graphics system might provide. The application of graphical techniques to flow chart programming has only been explored in a preliminary fashion thus far, but enough concepts and techniques have been developed to project the following description of the operation of a flow chart compiler. It is intended that the user would utilize a console equipped with display scope, light pen, keyboard, and typewriter. After calling up the flow chart compiling system the user might start, for instance, constructing the global flow chart of an electrical circuit simulator. Typically, he would call up an empty flow block, label it CIRCUIT SIMULATOR, then request from the library a flow block entitled GRAPHICAL DISPLAY AND CONSTRUCTION ROUTINE and connect this block's data tie to a blank data block which he labels CIRCUIT LIST STRUCTURE. After he connects the simulator block to the data block he would be through with the basic level and could continue by pointing at the undefined simulator block and activating an EXPAND key, thereby causing the display to change to an almost blank page with only a few data and flow connectors near the edge. Proceeding as before, it is possible to sketch out the rough flow structure of the circuit simulation technique and to continue to define the new blocks which have been introduced. In this way, he eventually arrives at a level where actual computation steps must be introduced into the flow blocks. Choosing an appropriate algebraic language, computation steps and conditional tests are entered via the keyboard. The algebraic statements appear on the display inside the designated flow blocks forming the primitive or atomic flow chart elements. Other places in the flow chart may require a list structure language instead of an algebraic one in order to move through the circuit elements in the CIRCUIT LIST STRUCTURE block. Making liberal use of the EXPAND and CONTRACT features, he is able to alternate between global and primitive structures as required by the problem. In this way, it is possible to make a rough layout of the flow and then proceed to work on details with the whole context in view.

During this phase of design, he is only concerned with the electrical simulation technique since the circuit will be created later by the Graphical Construction Service Program in the designated list structure. He need only specify what list blocks and ties are expected for each occurrence of the particular components which are defined. For output a library routine is called up which displays waveforms and is connected to the output data files.

Upon request to compile and run, the same display routines as were being used for flow charting are attached to the new simulator program, and the user begins to define the pictures which are to represent electrical components. The drafting section of the display routines permits the sketching of a graphical symbol which can then be identified with the appropriate electrical model by typing a previously specified identifier such as RESISTOR. As the symbols are called up and tied together with the light pen to form the electrical network, the display construction program creates the appropriate electrical list structure. Component values are entered by pointing to the component and depressing a key which causes the value to be displayed. Shaft encoder knobs are then used to adjust each value as desired. With the network designated and component values entered, another key is used to start the simulation. If a programming mistake is detected, error diagnostics are typed out and the flow chart of the simulator is displayed with an arrow pointing at the offending flow block. By using the EXPAND property, the precise error can be found, the necessary corrections made, and the simulation resumed. Once debugged and operating satisfactorily, the whole system is given a library name and as a service to future users of the simulator, operating instructions can be introduced into the computer as pictures.

Although a flow chart compiler is not yet available, a graphical circuit simulator similar to the one just described has been programmed for the TX-2 Computer at Lincoln and is almost ready to be used. This system should be capable of providing a circuit designer who has no programming experience, with a comprehensive simulation facility which he can use with little instruction to supplement his knowledge of the normal symbols, conventions, and graphical properties of electrical network theory. In cases where it is necessary to introduce non-linear elements the designer has the

ability to sketch in the detailed V-I characteristic. He can vary the values of components, gain of transistors, and modify the circuit topology as fast as he can sketch and operate the shaft encoder knobs.

In a sense the circuit designer must learn a type of programming language, but because it is graphical and problem-oriented it is natural and easy to learn. In order to realize a system of the kind described the user requires an on-line display scope with a fast response time from the computer and considerable computing power to accomplish the graphical manipulations within a reasonable interval. Thus, a large computer is required and to be economical it should be time-shared. The faster a machine can accomplish some computation such as rotating a circuit element in a drawing, the better the response time for the individual. Also the computer has more time left over for other users. Since it has usually been true that a bigger, faster machine has more computing power per dollar, it seems clear that machine size could continue to grow, and as long as the machine was efficiently time-shared, both the cost per user and the response time would be improved.

Display Consoles

It has become very clear from our experience on the TX-2 Computer, working with many graphical programs, that a useful display unit must do more than draw a single point for each data transfer. Vector drawing is essential and curve and character generation are almost as important. There are three reasons for these scope requirements. First, central computer time is too valuable to spend generating the long series of points or increments necessary for vectors or curves (using a separate smaller general purpose computer to do this work is bound to be more expensive than special purpose scope hardware). Second, the flicker rate on a 10 usec point scope is often excessive. Vectors can be drawn at 1 usec per point or faster. Third, the memory required for a point file, either in the central computer, or externally is too expensive and, in addition, the high speed transmission of point or increment information is very expensive if scope consoles are at all remote from the central machine. Complex vector type display units are necessary for graphics but, unfortunately, the commercial units produced so far have been very costly. It is possible, however, to time-share one

vector-curve generator for several display units if the generator is fast enough to eliminate excessive flicker. Each display then receives the common deflection signals, but is only intensified when the segments being drawn are intended for that unit. This technique allows the generator cost to be shared by up to four displays at present speeds. The generator may contain either analog or digital integration units. Analog systems are less expensive whereas digital units are inherently more accurate. Since, however, the accuracy of the display units themselves is limited, an analog generator can be constructed which appears to be comparable to a digital unit. Such a generator was recently built for the TX-2 to provide vector and curve drawing on several additional display consoles.

Memory for a vector display file requires about 4K words of storage per user. An optimal arrangement would be to have this memory available to the output channel as well as the computer and overlapped with the central computer memory so as not to waste processor time. It appears to be very advantageous to keep the display file as a list structure so that segments may be changed without rewriting the whole file. However, this means that the output channel should be capable of following list structures. Otherwise, as is current practice on TX-2, the central machine must interrupt its processing to do the list operations. This appears feasible on the TX-2 for perhaps four displays.

The hardware requirements discussed above are perhaps secondary to the software developments but still essential to achieving the full benefits of on-line graphical languages.

REFERENCES

1. Clark, W. A., et. al., "The Lincoln TX-2 Computer," Western Joint Computer Conf., p 143, (February 1957).

2. I. E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," AFIPS Conf. Proc. (1963 Spring Joint Computer Conf.), 23, p 329.

3. T. E. Johnson, "Sketchpad III: A Computer Program for Drawing in Three-Dimensions," AFIPS Conf. Proc. (1963 Spring Joint Computer Conf.), 23, p 347.

4. L. G. Roberts, "Machine Perception of Three-Dimensional Solids," Lincoln Laboratory Technical Report #315, (22 May 1963).

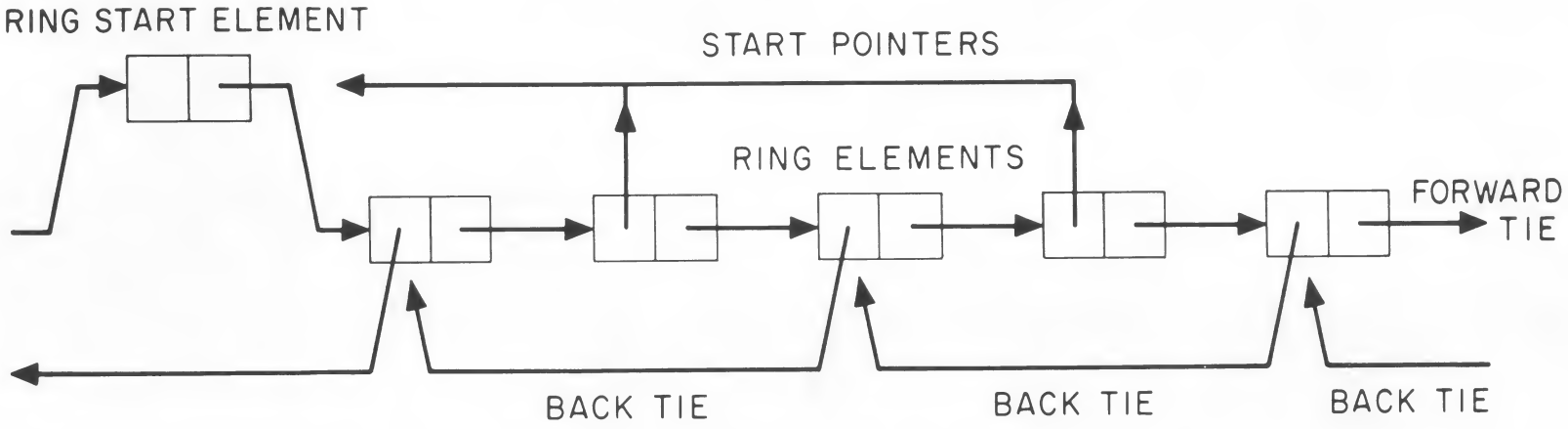

FIG 1 BASIC RING ELEMENT STRUCTURE

C23-356

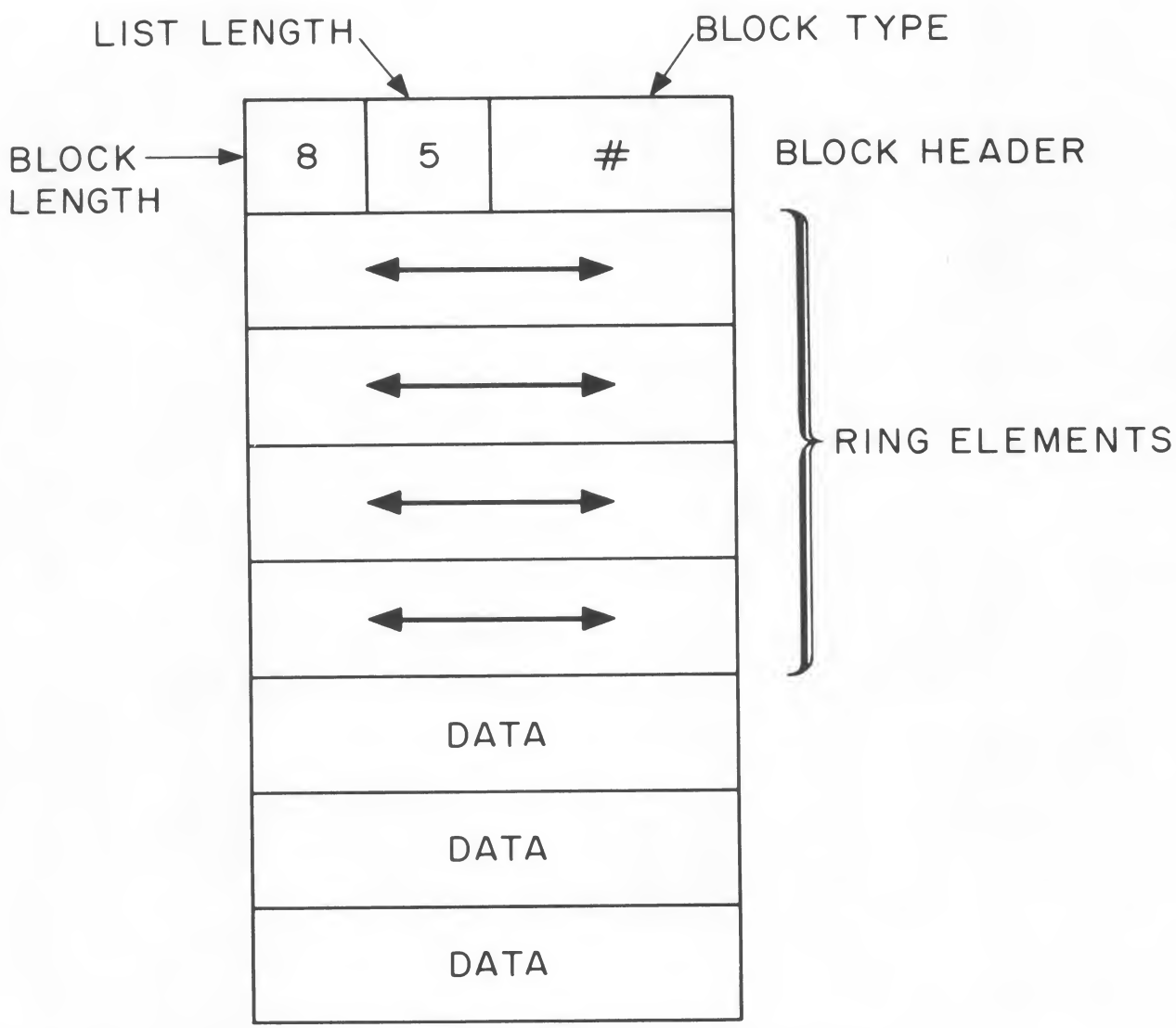

FIG 2 EXAMPLE OF BLOCK FORM

C23-357

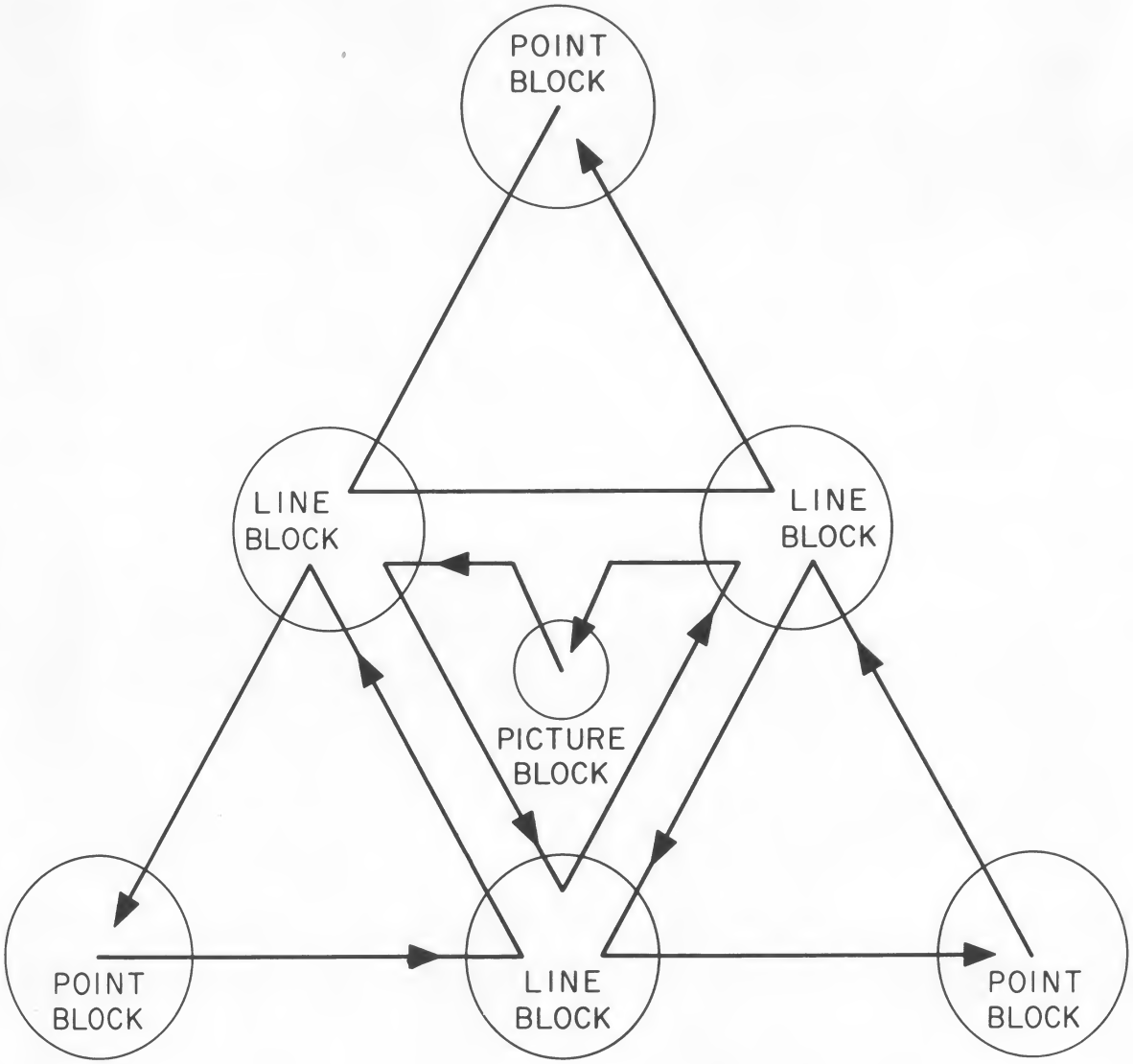

FIG 3 SCHEMATIC RING STRUCTURE FOR TRIANGLE

C23-358